National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r. 86/01)

Canada

0-315-23396-6

National Library of Canada
Canadian Theses Division
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Division des thèses canadiennes

302

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

Maylene Sharol McMillan

Date of Birth — Date de naissance	Country of Birth — Lieu de naissance
January 7, 1951	Canada

Permanent Address — Résidence fixe

11033 - 80th Avenue
Edmonton, Alberta
CANADA T6G 0R2

Title of Thesis — Titre de la thèse

Datatalk: A Data Management Facility for NIAL

University — Université

University of Alberta

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

Master of Science

Year this degree conferred — Année d'obtention de ce grade	Name of Supervisor — Nom du directeur de thèse
1985	William S. Adams

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date	Signature
August 2, 1985	Maylene McMillan

NL-91 (4/77)

The University of Alberta

DATATALK:
A DATA MANAGEMENT FACILITY FOR NIAL

by

© Maylene S. McMillan

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
FALL, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Maylene S. McMillan

TITLE OF THESIS: Datatalk: A Data Management Facility for NIAL

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1985

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Maylene McMillan

Permanent Address:

11033 - 80th Avenue
Edmonton, Alberta
Canada T6G 0R2

Dated 2 August 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Datatalk: A Data Management Facility for NIAL** submitted by **Maylene S. McMillan** in partial fulfillment of the requirements for the degree of **Master of Science.**

W S Hadaire
.....

Supervisor

Keith Amell
.....

In Amell Olsen
.....

S. J. Titus
.....

Date August 2, 1985

Dedication

This is for you Daddy.

Abstract

This paper describes Datatalk, a simple data management facility incorporated into the very high level programming language NIAL. The objective of Datatalk with NIAL is to make the data processing power of computers available to people who have small computational problems but who do not have the time or possibly the desire to master a conventional language. Many of these problems are handled very well by the numerous special purpose languages which have been developed for user-level programming. However, there are few alternatives available to the non-programmer who has processing requirements beyond the scope of the available packages.

Ideally, the end-user should have at his disposal both the capabilities of a data management system and an easy to use, general purpose language. It is shown that NIAL, a relatively new, interactive language is easy to learn and use, powerful, expressive and versatile, and is thus suitable for use by non-programmers. A data model based on the nested lists of NIAL is described. A data sublanguage supporting data definition, insertion, update, deletion and retrieval is added to NIAL. Comprehensive examples of the the use of Datatalk and NIAL are given.

Preface

NIAL terms and examples are in Helvetica throughout.

Definitions are introduced in Italics throughout.

Acknowledgements

I would like to thank my supervisor, W.S. Adams, for his patience, guidance and support throughout this research. I would like to thank my family for their encouragement and support.

Arthur Whitney deserves very special thanks. He contributed help in many ways.

Table of Contents

Chapter	Page
Chapter 1: INTRODUCTION	1
1.1. Overview with Example	1
1.2. Thesis Summary	7
Chapter 2: BACKGROUND	8
2.1. Motivation	9
2.2. Choosing a Language	11
2.2.1. Requirements	11
2.2.2. A Look at Some Languages	13
Chapter 3: THE NIAL LANGUAGE	17
3.1. Background - Array Theory	17
3.2. Language Overview	20
3.3. NIAL Characteristics	24
3.3.1. Interactive	25
3.3.2. Workspace Concept	25
3.3.3. Data Types	26
3.3.4. High Level Data Structures and Aggregate Operators	26
3.3.5. Picture Mechanism	27
3.3.6. Theoretical Background	27
3.3.7. Expressive Vocabulary	28
3.3.8. Extensibility	28

3.3.9. Help Facilities 29

Chapter 4: DATATALK DATA MODEL 30

4.1. NIAL Nested Lists 30

4.1.1. Nested List Representation 31

4.1.2. Strand and List Notations 32

4.1.3. Accessing Lists 33

4.1.4. Modifying Lists 34

4.2. Datatalk Tree 35

4.2.1. Datatalk Template 36

4.2.2. Accessing Datatalk Data 38

4.2.3. Modifying Datatalk Data 41

Chapter 5: DATATALK EXAMPLE 43

Chapter 6: CONCLUSION 55

6.1. Observations 55

6.2. Extensions 57

References 59

Appendix A1: DATATALK TERMINOLOGY 63

Appendix A2: REFERENCE MANUAL 65

2.1. Overview 65

2.2. General Notes 66

2.3. Data Description 66

2.4. Data Modification 67

2.5. Data Retrieval 68

2.6. Other 68

Appendix A3: DATATALK UTILITIES 69

3.1. Reporting 69

3.2. Searching/Selecting 69

3.3. Sorting 70

3.4. Analysis 70

Appendix A4: DATATALK PROGRAMS 71

Chapter 1

INTRODUCTION

Datatalk is an easy-to-use data management facility incorporated into the very high level programming language NIAL [Mor81]. This provides the user with an integrated facility for data description, insertion, update, deletion, retrieval and analysis - NIAL itself providing the last. The objective of Datatalk with NIAL is to make the data processing power of computers easily available to people who have small computational problems.

In the first section of this chapter a simple example of Datatalk in action is presented, accompanied by a general overview of the system. The second section outlines the organization of the remainder of this thesis.

1.1. Overview with Example

Datatalk is interpretive and interactive. Users work in a single, self-contained environment which contains the description of their data, the data, and any tools which they have assembled to work with the data, prepared by themselves or others. A copy of this environment may be saved at any time to be returned to later. A user may have any number of different Datatalk environments. On-line help facilities are provided which describe both Datatalk and NIAL operations.

To work with Datatalk and NIAL, users need only have a familiarity with their data and problem area and a rudimentary understanding of NIAL's mathematical concepts. However, the full power of NIAL is always available and it is expected that as users want to do more sophisticated things they will want to learn and use more of the expressive power of the

language. This thesis shows how Datatalk makes the learning process enjoyable and simple.

In the following example the user, MAY, wishes to store some information about a list of employees (staff). The data include a name, department number and phone number for each employee. After describing and entering the data the example proceeds with some simple reporting and analysis. Lines which begin with a number sign (#) are comments.

```
Q'Nial: UNIX Release 1 Version 3.03 Dec 6, 1983
clear workspace

      load "Datatalk
     Datatalk

          Welcome to Datatalk

What would you like to be called? May
What is the name of your data? Staff

What does a STAFF have? Employees
Is EMPLOYEES a list of similar items? yes

STAFF:
┌──────────────────────────────────┐
│EMPLOYEE EMPLOYEE EMPLOYEE ...    │
└──────────────────────────────────┘
Does this structure look right? yes

What does an EMPLOYEE have? Name, Dept and Phone

EMPLOYEE:
NAME DEPT PHONE
Does this structure look right? yes

What does a DEPT have? quit

MAY: # currently have an empty list of employees
     Staff
```

```
MAY: tally Employees

0

MAY: # assisted entry of data
     enter Employees

EMPLOYEE 0:
--NAME: ?  Doug
--DEPT: ?  23
--PHONE: ?  345

Would you like to add an EMPLOYEE? yes

EMPLOYEE 1:
--NAME: ?  Sue
--DEPT: ?  25
--PHONE: ?  432

Would you like to add an EMPLOYEE? yes

EMPLOYEE 2:
--NAME: ?  Bob
--DEPT: ?  23
--PHONE: ?  312

Would you like to add an EMPLOYEE? yes

EMPLOYEE 3:
--NAME: ?  Barb
--DEPT: ?  23
--PHONE: ?  320

Would you like to add an EMPLOYEE? no

MAY: # show the structure of Staff
     show Staff

STAFF:
```

EMPLOYEE EMPLOYEE EMPLOYEE ...

```
MAY: Staff
```

DOUG 23 345	SUE 25 432	BOB 23 312	BARB 23 320

```
MAY: # show the structure of an Employee
     show Employee

EMPLOYEE:
NAME DEPT PHONE

MAY: Employee Bob

BOB 23 312

MAY: Employees
```

DOUG 23 345	SUE 25 432	BOB 23 312	BARB 23 320

```
MAY: # some NIAL to find Dept 23 Employees
     23 findall Depts choose Employees
```

DOUG 23 345	BOB 23 312	BARB 23 320

```
MAY: Phonelist:= Name Phone of Employees
```

DOUG 345	SUE 432	BOB 312	BARB 320

```
MAY: # mix is a useful NIAL table primitive
     mix Phonelist

DOUG 345
SUE  432
BOB  312
BARB 320

MAY: # sort Employees by Names
     mix. gradeup Names choose Employees

BARB 23 320
BOB  23 312
DOUG 23 345
SUE  25 432

MAY: # functional programming
     # shows the power of NIAL
     sort_by is mix choose[gradeup second,first]
```

```
MAY: Employees sort_by Phones

BOB   23 312
BARB  23 320
DOUG  23 345
SUE   25 432

MAY: # when modifying, previous values are shown
     # (hitting return will leave as is)
     modify Dept of Employee Barb

DEPT: 23   24

MAY: Employees sort_by Depts

DOUG 23 345
BOB  23 312
BARB 24 320
SUE  25 432

MAY: enter Employees

EMPLOYEE 4:
--NAME: ?  Ken
--DEPT: ?  24
--PHONE: ?  440

Would you like to add an EMPLOYEE? yes

EMPLOYEE 5:
--NAME: ?  Rose
--DEPT: ?  25
--PHONE: ?

Would you like to add an EMPLOYEE? no

MAY: mix Employees

DOUG 23 345
SUE  25 432
BOB  23 312
BARB 24 320
KEN  24 440
ROSE 25 ?

MAY: # direct modification
     Phone of Employee Rose:=408
```

```
MAY: Employees sort_by Phones

BOB   23 312
BARB  24 320
DOUG  23 345
ROSE  25 408
SUE   25 432
KEN   24 440

MAY: delete Employee Bob

MAY: Employees sort_by Depts

DOUG 23 345
BARB 24 320
KEN  24 440
SUE  25 432
ROSE 25 408

MAY: [min,max] Phones

320 440

MAY: cull Depts

23 25 24

MAY: [min,max] Names

BARB SUE

MAY: stop
     bye
```

The above example contains data description, entry, retrieval and analysis. Describing the data is simply a matter of saying that a **Staff** has **Employees** and that an **Employee** has a **Name**, **Dept** and **Phone**. All data entered become part of one large NIAL data object. Subsets of the data are identified and extracted from this larger structure merely by providing the names used in their description. Useful NIAL primitives (like **mix**) can then transform the data into reports etc.. The sorting example is a use of the functional programming capability of NIAL and as such is moderately advanced. It was included to indicate the advantages of continuing education.

1.2. Thesis Summary

Chapter Two deals with the background and motivation for this work. Chapter Three talks about some basic NIAL concepts. Chapter Four discusses the Datatalk data model and Chapter Five provides a more extended example.

Chapter 2

BACKGROUND

Development of the microcomputer in the late 1970s brought a fast and dramatic drop in the price of computer power. More people began to consider them as a means of information storage and as a computational tool. The gap between users of computer power and computers has traditionally been filled by the professional analyst. Unfortunately, the demand for computer applications has expanded much faster than the supply of analysts and programmers needed to create them. Backlogs of two to three years are common in many companies. This is the so-called software crisis - the failure of software development to keep pace with hardware improvements. One approach to attacking this problem is to create tools which permit the ultimate users of computer power, the businessmen, accountants, scientists, engineers, secretaries and other nonprogrammer professionals, to carry out common tasks directly, without the intervention of a programmer. Datatalk with NIAL is intended to be such a tool. It is a data management facility embedded in a general purpose language. This chapter reviews the motivation for having such a facility and then looks at some general purpose languages which might have been used.

2.1. Motivation

The need for user-level programming has in part been met by query languages, database managers, spreadsheet calculators, business graphics and report generators, analysis programs, and systems which integrate some or all of these applications. Many routine data processing tasks are handled very well by these special purpose tools. In general they are easy to learn and use. Ease of use is achieved by providing users with a single environment, an interactive interface, a simple and fault tolerant syntax, meaningful error messages, help and possibly tutorial facilities. Several of the most popular systems have a very strong visual component. Query-by-Example [Rei81] uses tables and spreadsheets use grids. The users can see the organization of and relationships among their data. User-oriented languages are described as being "nonprocedural" to some degree - that is, they allow the user to specify what has to be done, rather than specifying the procedure by which it is done. This is achieved by automating certain functions and details which are common to the application area of the program. Generally, the smaller the application domain the easier this is to do. However, in the attempt to simplify some aspect of programming, power and versatility have been restricted. For one-shot problems, or problems which are beyond the scope of the available packages, the user often turns to a general purpose programming language.

Conventional programming languages are in general not suitable for use as end-user problem solving tools. The disparity between the level of discourse afforded by these languages and the user's conceptualization of his problem is too great. Conventional languages reflect very closely

conventional computer architectures. Data structures, operations and control structures provided for the user mimic the very low level on which data items are actually stored and manipulated. The majority of applications deal with collections of data objects that are far removed from this simple computer model. The data structuring facilities provided by most languages, such as arrays and files, are primarily implementation abstractions. They do not support abstractions which are useful in understanding real world data relationships. The user must select representations for the naturally occurring data structures of the problem environment, and must effect his solution in terms of these representations. This often involves specifying numerous details which are necessary for the representation but extraneous to the solution itself. Moreover, any data which are to outlive programs must be stored in files separate from the programs which deal with them. These files have no structure apart from that provided by the programs. The only opportunity to view the data in a meaningful way is to provide detailed printing and formatting commands to display them.

Ideally, an end-user should have at his disposal both the capabilities of a data management system and an easy-to-use general purpose programming language. In Programming Language Standardization, [Oll80]." Dr. T. William Olle, a member of the ISO working group associated with database management predicts:

> "Although standardization in programming languages should settle down somewhat during the '80s, the most significant perturbation to the otherwise tranquil picture will be the infusion of database facilities into the programming languages. Database management can never be divorced from programming languages."

The work in this thesis incorporates database management as part of a programming language. The success of any undertaking in this area which hopes to be useful to nonexpert programmers depends to a great extent on the programming language chosen.

2.2. Choosing a Language

There is a multitude of general purpose programming languages in existence but only about a dozen of these are in widespread use. Each language has its own distinctive grammar and syntax; each has its own manner of expressing ideas. Writing a program for a given task would be easier with some languages than with others. Some languages are more suitable for use by end users than others. In the first part of this section the general characteristics desired of a user-oriented general purpose programming language and particular features of a language required for use with a data management system are presented. In the second part several languages which are widely used by nonexpert programmers and several others which more closely match the requirements are examined. One language, NIAL, is found to be particularly suitable.

2.2.1. Requirements

George Boole in his Laws of Thought [Boo51] asserted

> "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

Four characteristics which a computer language should embody if it is to be useful as a "tool of thought" [Ive80] are: simplicity, power, expressiveness

and versatility.

Simplicity refers to the ease with which a language can be learned within a short period, as least to the degree that a subset can be used to obtain useful results.

Power is the capacity to perform effectively. It is used here as a measure of the ease with which the language can be used to solve complicated problems.

Expressiveness is the quality of serving to suggest a representation of something and also of serving to symbolize something. Here it measures the extent to which the language facilitates thinking about various problems. The closer the means of expression are to the programmer's natural conceptualization of the application, the less mismatch there will be between what he says and what he means, between what he wants and what he gets.

Versatility is the capability of turning competently from one task to another; the quality of having generalized aptitude or varied uses, of serving many functions. A versatile language may be used for a variety of purposes.

In addition to these general characteristics, there are several particular features which are required of a language if it is to be used in an integrated fashion with data management facilities. Firstly, the language must be *interactive*. This is essential to its potential simplicity. The user must be able to view his data, to try out various operations on them and see immediately the effect. Secondly, the language must provide ways of processing entire data structures at once. Databases contain a high degree of regularity and may be conveniently manipulated by a set of *aggregate operators*. As with relational processing, the "primary purpose is loop-avoidance, an absolute

requirement for end users to be productive at all" [Cod82]. This capability should also be available for analysis of the data retrieved if the system is to be harmonious and easy to learn and use.

2.2.2. A Look at Some Languages

The general purpose language most commonly used by nonprogrammers at present is BASIC. It was developed in 1964 at Dartmouth College primarily as a language for introducing nonscience students to computing. Influenced by Fortran and Algol, BASIC was designed to meet the need for an easy-to-learn beginner's language that could work in a friendly nonfrustrating programming environment. The advent of personal computers pushed BASIC into a more extended role. It became the language chosen for use on them during the middle to late 70s because it was small enough to fit in the limited memory of the early machines and was considered easy to learn and use. BASIC has evolved into one of the most popular programming languages, available for every microcomputer, usually provided free, and often embedded in the ROM of the system. As a consequence of its use over the years, there are many books about it and a large number of programs available to users.

Its key feature, an important advance at the time, is its interactiveness. Successive lines of a program are typed directly to the BASIC system and the program can be run immediately. In addition, most systems can execute single lines of commands outside of formal program definitions. The language itself is simple. Variables are either numeric or alphanumeric, and the only structure to group several pieces of information is the array. Some of the

most difficult parts of Fortran were removed, obviating a detailed specification of the format of inputs and outputs.

The designers did achieve simplicity but the cost was power and expressiveness. The limited data structuring and manipulation facilities of BASIC make it quite unsuitable for our purposes. The only method of organizing data is the array which must be uniform in type. There are no operations which deal with arrays, so the user must provide all the logic to process the elements one at a time. Two of the most egregious flaws are the absence of subroutines or procedures and the lack of adequate control structures.

The second most commonly used language on microcomputers, and thus by many nonprogrammers, is Pascal. Developed in 1970, it was designed with three main objectives: support modern programming methodology, be simple enough for students to learn, and be easy to implement reliably, even on small computers. These goals have been achieved in that Pascal has been broadly adopted as an instructional language, and has been implemented on almost all computers. It has become the preferred language for use where BASIC is less than satisfactory - its structured programming capability makes it easier to work with for programs of any length or complexity.

In addition to the standard control constructs of structured programming, Pascal has quite powerful data structure facilities that are suitable for defining abstractions. A list of arbitrary constants may be defined as an enumerated type, heterogeneous records with individually named fields may be defined, a data structure may be named as a type, and data types can be dynamically allocated and referred to by pointers. However, users cannot really think in terms of whole abstract structures since processing of the data

must take place at a lower level of abstraction where all operations are broken down into steps that manipulate the contents of single memory locations in the computer.

The major drawback to Pascal for our purposes is that it was developed for a batch environment. Such an environment introduces barriers between the programmer and his programs and data. The program may be in one file, the compiled program in another, library functions in another, data in still another. The user has to cope with several different environments, dealing with an editor, compiler, linker, operating system and programming language, each having a different syntax and semantics. Experiments with one or two commands consume disproportionately large amounts of time and effort. Moreover, there is no way to store a data description apart from programs that manipulate the data.

Another class of languages has evolved in an attempt to address the limitations in power and expressiveness of conventional "high level" languages. These have been termed ***very high level languages (VHLLs)***. The intent is to make it possible for the programmer to work at a higher level of abstraction. This should make his task easier and less error-prone since he doesn't have to deal with computer-related details inessential to his problem. Basic features of VHLLs are implicit specification of control flow, high-level data structures and associated aggregate operators, and associated data referencing.

There are a number of "general purpose" very high level languages which are intended to have application to a broad range of programming problems. This group includes SETL [DSS83], LISP [MAE65] and APL [Ive62]. APL and LISP are highly interactive, and are supported by complete self-contained

programming environments, which include an editor, interpreter, and coordinated facilities for memory management, error recovery and I/0 formatting defaults. Both provide aggregate data structures (lists, tables and arrays of higher dimension in APL and lists, the elements of which may themselves be lists, in LISP), and operations for creating and processing entire structures at once. As well they have a functional notation that encourages a modular programming style and often emphasizes the hierarchical structure of a computation. While both languages are extremely powerful and expressive, many users believe that their unusual notations will make them difficult to learn and use. APL employs special characters for operations and requires terminals outfitted with APL keyboards. LISP uses reverse Polish notation and often requires many parentheses to delineate the structure of a computation.

NIAL, a relatively new very high level language combines LISP and APL concepts, while avoiding the notational difficulties, and provides structured programming features found in languages like Pascal. NIAL embodies the characteristics of simplicity, power, expressiveness and versatility desired in a user-oriented general purpose programming language. The language and its suitability for the purposes of this thesis are described in Chapter 3.

Chapter 3

THE NIAL LANGUAGE

The Nested Interactive Array Language, or NIAL as it is usually called, has been designed jointly by M.A. Jenkins of Queen's University at Kingston, and Trenchard More of the IBM Cambridge Scientific Centre [Mor81]. Q'NIAL is a portable implementation of the language which has been developed at Queen's University [Jen82]. It has been implemented on many computers including large time sharing machines and 16-bit personal computers [Sut83].

NIAL is an interactive general-purpose programming system. *Array Theory* serves as the model for the data manipulated in the system and provides the definitions for the data operations of the language. The first part of this section contains a brief description of NIAL's theoretical background in array theory. The second provides a brief overview of the language. In the third part those features of NIAL which make it suitable for use with a system like Datatalk are examined.

For further information on NIAL the reader is referred to [Mor81] [Jen82] [ScJ82a] [Ada83] [Sut83] [Jen84].

3.1. Background - Array Theory

The development of array theory began in the late sixties with the work of Trenchard More. The fundamental objective was to create a mathematical foundation for the manipulation of data objects in computers. Objects considered in data processing such as numbers, characters and indivisible words often occur as collections, such as lists, tables and databases. Collections are

frequently nested in the sense that some or all of the objects of a collection may themselves be collections. Models of data used in linear algebra, set theory, APL, LISP, and relational and hierarchical databases are all collections in this sense. When stored or communicated by physical devices, collections become ordered in time or space and their items are often placed in rectangular arrangement. Finite, rectangularly arranged, nested collections (*arrays*), represent quite directly the common kinds of objects and collections that constitute data.

Set theory is the study of the nested set as a mathematical model of concepts. Since mathematics is largely concerned with the study of the infinite and language enables us to reason about infinite collections, sets are abstracted from language. On the other hand, physical objects and collections are finite, and arrays are abstracted from spatial arrangements of physical objects. They have the properties (lacking in sets) of order, repetition, type and multiple axes, and reflect one's geometric and pictorial perception of collections much more closely than sets. However, More has shown that the principles of nested collections developed in set theory apply with few changes to the nesting of arrays [Mor81].

Array theory considers a universe of objects together with certain agents that act on the objects. The objects are called *arrays*. Since they are the only kind of object to be observed directly in the universe of discourse, the theory is said to be *one-sorted*. The agents are called *operations*. They are observed indirectly by the effect they have on arrays. The result of applying any array-theoretic operation to an array is again an array because the theory is one-sorted. The advantage of one-sortedness is simplicity. It is not necessary

to restrict domains of validity to particular universes when making assertions about objects and about relationships between operations applied to objects. One-sortedness implies closure. The universe of arrays is *closed* in the sense that all operations of the theory, when applied to objects in the universe, result in objects again in the universe. The advantage of closure is that the result of any operation may be used as the argument for any other operation.

Only two types of primitive objects are dealt with by pure array theory - ordinal numbers and Boolean truth values. An adjunct theory that includes several more types serves as the basis of the NIAL language and system. Eight primitive operations were chosen for the pure theory; from them are defined all of the other basic operations that are added to the pure and applied theories for the sake of convenience and clarity. The basic operations are total in the sense that each of these operations is defined for all arrays taken as arguments. The operations combine coherently in a substantial algebra that permits the creation of many identities and equations. These can be used to test the consistency of the theory, to simplify and prove the equivalence of definitions and to verify the correctness of an implementation. The principal finding of array theory is that most structural facts about ordinary arrays continue to hold unchanged at the boundaries of emptiness, singularity and nesting. The theory insures that constructions designed for the ordinary case extrapolate reliably to boundary cases - any operation on any array has a well-defined result.

3.2. Language Overview

NIAL is intended to be used as a high level design tool. It was implemented to utilize the powerful processors and large memories that are becoming available for personal computers and workstations. The goal was to have it usable by people with no prior programming experience and with only a rudimentary understanding of mathematical concepts.

NIAL is an interactive language. It is expression-based in that the principal unit of computation is an expression that returns a value. This value is always an array, which is displayed as a "picture" showing its nested structure and content. In order to make it easy to use NIAL in an imperative (command oriented) way, a syntactic means has been provided to suppress the result of an expression. Program texts can be read from files or loaded as predefined operations. NIAL has a flexible definition and expression syntax that supports a wide range of programming styles. A block structured procedural style, a strictly functional style or a combination may be used.

NIAL data objects are nested rectangular arrays with 0, 1, 2 or more dimensions. Each location in an array holds another array. The nesting terminates with atomic arrays . An *atom* is a non-divisible object that contains itself as its sole item. There are seven types of atomic arrays in NIAL: truth values, integers, real numbers, complex numbers, characters, phrases and faults. Arrays may contain any combination of atomic types, permitting the aggregation of data into structures that capture the relationships among data items. NIAL uses a picture mechanism which draws frames around the items of complex arrays in order to assist visual comprehension of their structure. Two levels of structural detail may be specified. The arrays in the examples

in this chapter are displayed in *sketch* mode as are all examples in the thesis. The following are examples of arrays as they are displayed by NIAL.

```
7
o 7 7.7 7.7j4.5 'm "myphrase ?myfault
```

0	0	0	1	0	2
1	0	1	1	1	2

The first example, the integer 7, is an atomic array. The second example is a one dimensional array or list containing seven items, each one representative of a different atomic type. The third example is a two dimensional array or table with two rows and three columns. Each item in this table is a pair (list of length 2).

An *operation* is a NIAL object which, when applied to an array, yields a new array; a *transformer* is an object which, when applied to an operation, yields a new operation. An *expression* is any valid Nial construct that describes a value (and thus yields an array when evaluated). NIAL provides many predefined operations and transformers as well as a mechanism for defining new ones. Many of NIAL's arithmetic and array manipulation operations are similar to those found in APL. A large group of operations are devoted to building and modifying arrays, and a subset of these are similar to the list features found in LISP.

The syntax of NIAL supports both prefix and infix notation for operations. For example,

2 plus 3 is equivalent to **plus 2 3**.

The prefix notation is the basic syntax. Infix always causes the creation of a pair to be passed to the operation. Thus any infix can be expressed as prefix. In general,

A operation B is equivalent to *operation A B.*

Lists of data can be generated by placing the data side by side. Parentheses are strictly reserved as grouping symbols. Another way to generate lists of data or lists of operations is to enclose the objects with brackets and separate them by commas. These notations are described in detail in chapter 4. Here are some examples:

```
      Names:= "Albi "Marcia "Sue "Rose
Albi Marcia Sue Rose

      Ages:= 32 30 29 33
32 30 29 33

      Numbers:="433-6905 "433-5066 "451-2018 "488-3415
433-6905 433-5066 451-2018 488-3415

      type Names
" " " "

      type Ages
0 0 0 0

      tally Names
4

      sum Ages
124

      tally Ages
4

      [sum Ages, tally Ages]
124 4

      [sum, tally] Ages
124 4

      124 div 4
31.
```

```
      div 124 4
31.

      div [sum,tally] Ages
31.

      average is div[sum,tally]

      average Ages
31.

      Names append "Kyla
Albi Marcia Sue Rose Kyla

      EACH first Names Ages Numbers
Albi 32 433-6905

      Names:= rest Names
Marcia Sue Rose

      Numbers:= rest Numbers
433-5066 451-2018 488-3415

      Ages:= rest Ages
30 29 33

      "Marcia find Names
0

      0 pick Numbers
433-5066

      "Rose find Names pick Numbers
488-3415

      findnumber is op name\
      [name find Names pick Numbers]

      findnumber "Sue
451-2018

      Ages >= 30
101

      Ages >= 30 sublist Names
Marcia Rose

      "Albi "Rose EACHLEFT in Names
01
```

```
      Names Numbers
┌──────────────┬──────────────────────────┐
│Marcia Sue Rose│433-5066 451-2018 488-3415│
└──────────────┴──────────────────────────┘

      flip Names Numbers
┌──────────────┬──────────────┬──────────────┐
│Marcia 433-5066│Sue 451-2018  │Rose 488-3415 │
└──────────────┴──────────────┴──────────────┘

      mix flip Names Numbers
Marcia 433-5066
Sue    451-2018
Rose   488-3415
```

NIAL execution occurs in a workspace that contains both system and user-defined data, expressions, operations and transformers. Workspaces may be saved and restored. Built-in system operations for input and output provide for access to data outside the workspace. All editing of program text is achieved by an interface to the host system.

NIAL is extensible in that the operations and transformers created by the user may be utilized exactly the same way as predefined ones. The new definitions are created incrementally either in successive interactions with the user or by reading from a file of definitions.

Debugging is aided by the ability to set trace and breakpoints in functions.

3.3. NIAL Characteristics

This section reviews the characteristics of NIAL which provide the simplicity, power, expressiveness and versatility that make it an ideal language for supporting a system like Datatalk.

3.3.1. Interactive

A high degree of interactiveness contributes significantly to the ease of use of NIAL. The user may type in a single expression, and the computer instantly supplies a response. Sample data can be quickly and easily created. Thus it is very easy to experiment with various operations and combinations of operations. Since results are seen directly and immediately, the learning process is facilitated. This interactive environment, complemented by the functional nature of NIAL, encourages a modular programming style in which a problem is broken down into smaller pieces and a short function created to accomplish each one. A function may be built up piece by piece by combining expressions. Each expression and function can be created and tested separately and immediately, so logic errors are relatively easy to detect and rectify. A modular approach helps to reduce the complexity of problem solving by reducing the amount of information and detail which must be dealt with at one time.

3.3.2. Workspace Concept

The workspace concept provides users with an easily-grasped mental model of the environment in which they are working. All the data and functions required for a particular problem can be assembled and used in one place, and are instantly accessible by name. This allows users to work in a natural way - to look at their data, formulate hypotheses and immediately test them, without the distractions caused by moving from file to file to examine and edit procedures, data and output. At any time the entire active workspace may be saved in the state that it is in and later retrieved.

3.3.3. Data Types

There are no type declarations in NIAL. A variety of data types are offered corresponding to the kinds of data values most often used. The same operation may be applied to data of any type for which it makes sense. Users may treat words as units rather than as strings of characters. The *fault* data type is particularly useful in database applications and provides a very good solution to the problem of representing missing data. (For example, see the processing involving the missing **Price** in Chapter 5.) Values of any type may be freely intermixed in arrays and every data object "knows" the type of the values it contains. Users need not be concerned about details of formatting, as this is handled automatically by the language processor.

3.3.4. High Level Data Structures and Aggregate Operators

The nested array is a data model which permits aggregation of data into structures that directly capture the relationships between data items. This makes it possible to conceptualize, organize and express a program in terms of natural data objects. Many operations are provided to manipulate these aggregate data structures, so that any solution representation which makes use of them does not have to supply information on how to use them. They may be constructed, accessed and utilized in a unitary fashion. Freed from the distracting details involved in item by item processing, which may not be explicitly required by the problem, the user is allowed to focus on his intent rather than on the means by which this intent is to be realized.

It is possible to identify and extract particular objects from these high-level data structures by describing the objects desired in terms of their values

or other properties. It is not necessary to provide an algorithm for locating them. Again, this allows much low-level detail to be elided.

3.3.5. Picture Mechanism

Nested box diagrams are used to show the structure of arrays. Diagrams are effective in representing the organization of data and are about as close as the user can get to seeing the invisible objects the computer holds in its memory. (There is a precisely defined algorithmic relationship between the internal data object and its two-dimensional picture on the screen [ScJ82b]). They also help the user visualize how to represent the data encountered in a problem. Diagrams are useful in communicating concepts about structural transformations and combinations of data; they provide the user with visual cues for actions.

3.3.6. Theoretical Background

The mathematical nature of array theory provides a sound theoretical basis for NIAL and contributes to the simplicity and ease of use of the language. The definitions of array theory operations are intended to be exception-free. Constructions designed for ordinary nested arrays extrapolate reliably to the boundaries where nesting terminates or where arrays become empty or singular. The user need not be concerned with these special cases - considerations that are a major source of error and consume a disproportionate amount of time. Every operation and combination of operations has a predictable result. Machine level errors are handled in a reasonable way. A perfectly viable data item is produced, a *fault*, which indicates the kind of error and propagates through subsequent transformations like any other data

type. This allows errors to percolate up in a controlled fashion. Emptiness can be dealt with and represented in a natural manner - for every unique array there is an empty one which corresponds to it, has the same shape and type, and can be manipulated in the same way.

3.3.7. Expressive Vocabulary

The closer the means of expression of a language are to a person's natural ways of thinking and working, the closer the match will be between what he says and what he means, between what he wants and what he gets. NIAL uses ordinary words for the operations of array theory. These words and the data handling actions they represent reflect quite directly the way the solution to a given problem might be expressed in natural language. The gap between thought and implementation is minimized. Typing at the terminal is simplified by the fact that capital letters and the corresponding lower case Roman letters are interchangeable.

3.3.8. Extensibility

NIAL may be easily customized for an application by adding to the initial general purpose vocabulary. Its definition mechanism provides the means for users to give their own names to basic operations and combinations of operations, as well as to create new operations and transformers, all of which may be utilized in exactly the same way as predefined ones. Using a problem-specific vocabulary and the functional programming tools of NIAL (composition and transforms) to build up a collection of operations, expressions can be made to read almost like natural language. This allows programs to be written in such a way that they are comprehensible to and usable by almost

anyone familiar with the application area. The extensible nature of NIAL also facilitates a modular programming style.

3.3.9. Help Facilities

There is an on-line help facility which contains information on the operations, syntax and semantics of NIAL. A library of additional operations which may be useful to the programmer is also provided.

Chapter 4

DATATALK DATA MODEL

The data model used for Datatalk is based directly on the nested lists of NIAL. In defining the structure or form of his data the user automatically defines a nested list template or schema (intension). This is called the Datatalk Template. His data are always stored in an actual nested list (extension) which matches the current template. This is called the Datatalk Data. In defining or redefining the structure of his data the user is changing the Datatalk Template. In entering or modifying his data the user is changing the Datatalk Data. The first section of this chapter reviews the generation, accessing and modifying of NIAL nested lists. In the second section the analogous concepts of Datatalk are presented.

As seen below, nested lists can be represented by trees. Datatalk's essential addition to NIAL is to allow the accessing of subtrees by name. A more formal description of the data model is given in Appendix A1.

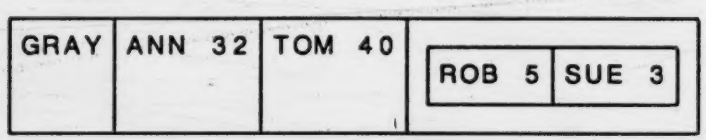
4.1. NIAL Nested Lists

A *list* is an ordered collection of zero or more objects, located at addressable positions along one axis. As these objects may again be lists, a nested data structure is defined. These addressable objects are the *items*. The **tally** of a list is the number of items in the list. A list for which the tally is 0 is an **empty** list. Items can be addressed by an integer index (origin 0). The most deeply nested objects are called the *leaves* of the list. A nested list can be represented by a *tree*, i.e. a node for a list and a son node for every item of a list. Thus leaves or empty lists have no sons. A subtree is the tree itself or an

item of a subtree. Any subtree can be accessed via a *path*. A path is just a list of successive addresses.

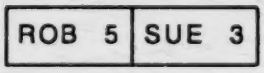
4.1.1. Nested List Representation

A NIAL list is displayed by means of a rectangular box notation. Consider the following list containing data about the Gray family:



This list has 4 items: family name, mother, father and children. The **first** item is a leaf. The **second** and **third** items are each lists with 2 items: name and age. Name and age are leaves. The **last** item is a list of 2 children. Each child is a list with 2 items: name and age. In NIAL terms (if this list were called Family_Gray):

```
      tally Family_Gray
4

      first Family_Gray
GRAY

      second Family_Gray
ANN 32

      third Family_Gray
TOM 40

      last Family_Gray
```

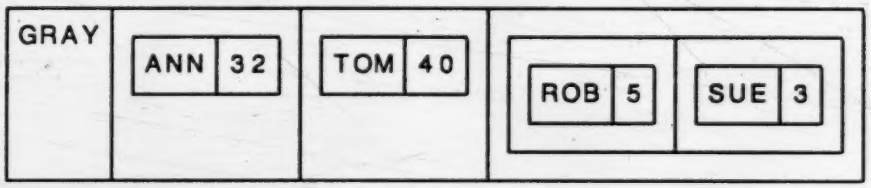



```
      EACH tally Family_Gray
1 2 2 2
```

```
      EACH tally last Family_Gray
2 2
```

The rectangular notation used by NIAL for the display of arrays employs a partitioned box to represent lists, with a cell for each item of a list. This notation is used recursively for sublists. There are two modes of display in NIAL: *sketch* and *diagram*. The above examples have been done in sketch mode; when all items of a list are leaves then the partitioned boxes are omitted. They are included in diagram mode. For example:

```
      set "diagram
sketch
      Family_Gray
```



4.1.2. Strand and List Notations

In NIAL, two or more data objects placed side by side on a line of input are collected into a list with as many items as there are data objects. This process is called *strand notation*. Parentheses are used as grouping (nesting) symbols. For example, Family_Gray is equivalent to the following strand:

```
"GRAY ("ANN 32) ("TOM 40) (("ROB 5)("SUE 3))
```

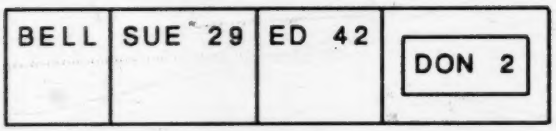
Another way to generate lists is via *list notation*. Here the list is enclosed in brackets with items separated by commas. This notation can nest (or mix with strand notation). With list notation it is possible to generate lists with only 1 item or even the empty list: [ ]. Thus any strand can be captured with

list notation but not vice-versa. Family_Gray is equivalent to the following list:

```
["GRAY,["ANN,32],["TOM,40],[["ROB,5],["SUE,3]]]
```

The following mix of strand and list notation generates another family. Note that the list notation is necessary to capture the idea that this family has a list of only one child:

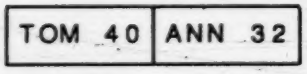
```
Family_Bell:=["BELL,"SUE 29,"ED 42,["DON 2]]
```



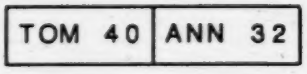
4.1.3. Accessing Lists

There are three ways to select data from a nested list: **pick** - to pick out an item, **choose** - to choose a list of items, and **reach** - to reach down into the tree to some subtree. The address (for pick), addresses (for choose) and path (for reach) are 0-origin integer indices, i.e. 0 corresponds to first, 1 to second, etc.. The examples in this section use the lists of names and ages introduced in the previous sections:

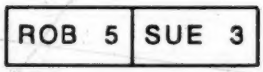
```
      0 pick Family_Gray
GRAY
      [2 pick Family_Gray,1 pick Family_Gray]
```




```
      2 1 choose Family_Gray
```



```
      3 pick Family_Gray
```

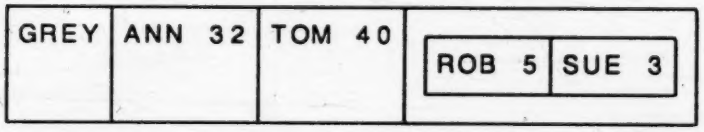



```
      1 pick(3 pick Family_Gray)
SUE 3
      0 pick(1 pick(3 pick Family_Gray))
SUE
      3 1 0 reach Family_Gray
SUE
```

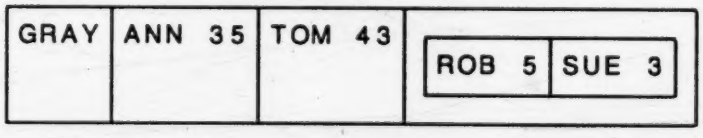
4.1.4. Modifying Lists

The three ways for changing the data in a list that correspond to the three ways of accessing data are: **place**, **placeall** and **deepplace**:

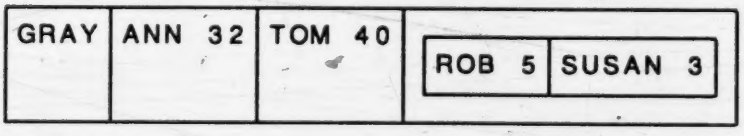
```
      "GREY 0 place Family_Gray
```




```
      ["TOM 43,"ANN 35] (2 1) placeall Family_Gray
```




```
      "SUSAN (3 1 0) deepplace Family_Gray
```



4.2. Datatalk Tree

The purpose of Datatalk is to allow users to describe the structure of their data and to allow them to access their data with the structure names used in the description. This obviates the need for remembering and using integer position indices.

When describing data, there are three kinds of abstractions commonly used in natural language to understand and simplify the expression of real world relationships. The first is *aggregation*, which allows a relationship between named objects to be thought of as a (higher level) named object. For example, the relationships among a name, mother, father and children may be abstracted as the aggregate object "family". The name "family" may be used without bringing to mind all the details of the underlying relationship. Another kind of abstraction is *generalization*, in which a set of similar objects is abstracted as a single generic object. For example, each "family" has the same kind of information associated with it, as does each "mother", "father" and "child". The third abstraction is the idea of *collection*, in which similar objects are grouped together. In this chapter's example there is a collection of families (currently 2) and each family has its own collection of children.

An aggregation (e.g. family) is a grouping together of diverse objects (name, father, mother and children) under a single name. A collection (e.g. children) is a grouping together of similar objects, i.e. objects of the same generic type (child). Collections may grow or shrink in number as individuals are added or deleted, e.g. a family may have more children. Aggregations are fixed in form, e.g. a family always has a name, mother, father and a list (possibly empty) of children.

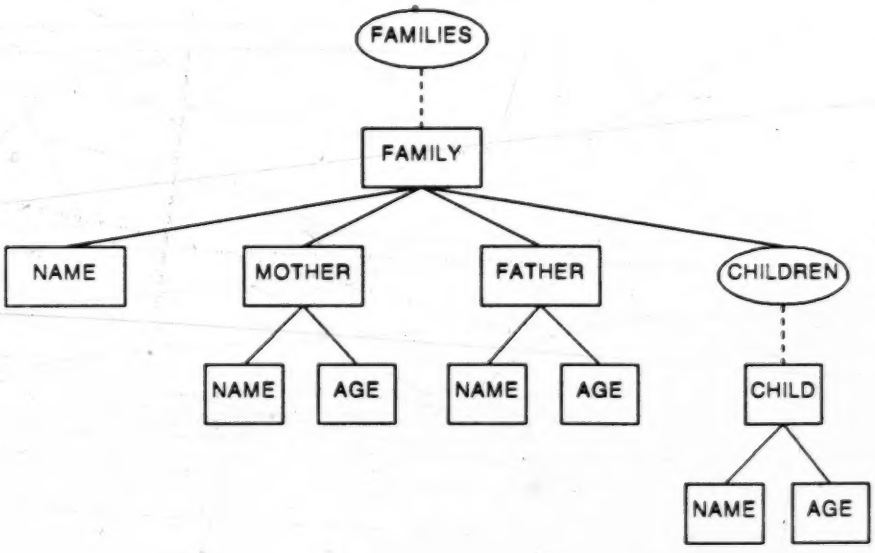
Datatalk enables the use of these abstractions in the description and manipulation of data. The following sections continue with the family example.

4.2.1. Datatalk Template

Consider the following Datatalk statements about the structure of a community:

```
Community has Families
Family has Name Mother Father Children
Mother Father Child have Name Age
```

From this description Datatalk constructs a template for the organization of the data to be stored. This template or schema may be represented by a tree:



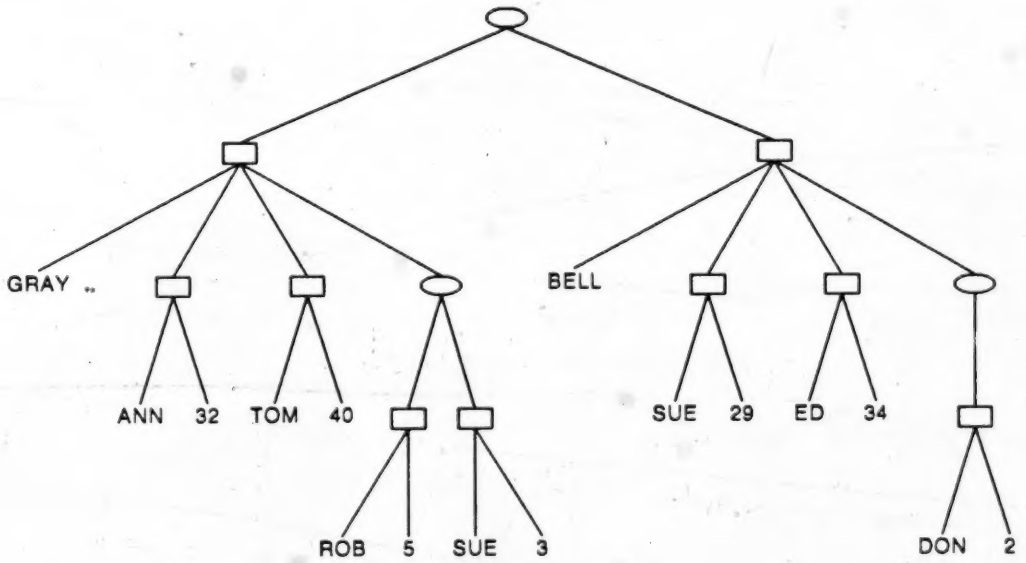
Generalization is reflected in the description of typical (generic) objects as shown in the tree. Aggregation is supported by providing the means to

name relationships among objects at various levels - that is, to name every node in the tree. Collections are provided by the ability to use plural names (e.g. **Families** and **Children**) when defining the structure. A *plural* node is drawn as an oval, with the associated name inside. This indicates that the user can have zero or more items (of similar structure) in the data at that location. This is depicted by an edge consisting of a broken line to one typical node of that kind. The data may grow and shrink at plural nodes. It is at these places that insertion and deletion are allowed, providing for a dynamic data structure. FAMILIES and CHILDREN are plural nodes in the example. A *singular* node is drawn as a rectangle, with the associated name inside. The name is an aggregation, naming the relationship among its component items, which in some sense are attributes of the object defined. A *leaf* node is a singular node with no sons, i.e. a terminal node. It names a primitive kind of data object. It is at these locations that the user's data are stored. NAME and AGE are examples of leaf nodes.

This data template is displayed by Datatalk as follows:

```
COMMUNITY:
┌──────────────────────────┐
│ FAMILY FAMILY FAMILY ... │
└──────────────────────────┘

FAMILY:
┌──────┬────────┬────────┬──────────────────────┐
│ NAME │ MOTHER │ FATHER │ CHILD CHILD CHILD ... │
└──────┴────────┴────────┴──────────────────────┘

MOTHER FATHER CHILD:
NAME AGE

NAME AGE:
leaf name
```

The family data of previous sections constitute a possible extension of the template (intension) described above. This will be the data assumed for the following sections. The collection of the two families is shown here as a tree:

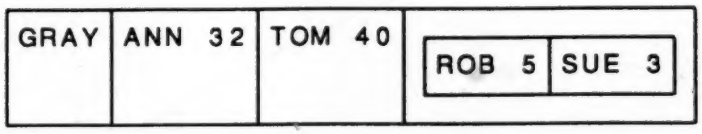


4.2.2. Accessing Datatalk Data

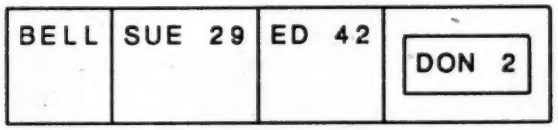
All data stored by Datatalk is organized according to a template such as the one in the previous section, created from the user's description of the data. One NIAL data object - a nested list - holds the data. There is a direct correspondence between the structure of this list and the structure of the template. Items of the list may be selected, again by providing a description of how to get to the particular data value. However, rather than providing a path consisting of integer addresses, the path is given in terms of the names used in the data description. When choosing particular items from a collec-

tion (plural node) either integer or name indices can be given (if NAME is one of the attributes). For example, **Family 0** is equivalent to **Family Gray** and **Family 1** is the same as **Family Bell**.

```
      Family Gray
```




```
      Family 1
```




```
      Mother of Family Bell
SUE 29

      Child Sue of Family Gray
SUE 3

      Age of Mother of Family Bell
29

      Age of Child Sue of Family Gray
3
```

Every structural name has a singular and a plural form, e.g. Mother-Mothers, Family-Families, Child-Children. The examples above use only singulars. As such they form paths which specify a single subtree. This corresponds exactly to NIAL **reach**, i.e. at every level of the tree a **pick** is done. A plural selects the list of all subtrees in the data whose structure corresponds to the singular of that name. For example, **Mothers** will select the data associated with every mother, and **Children** will select every child.

Mothers

ANN 32	SUE 29

Children

ROB 5	SUE 3	DON 2

Children of Family Gray

ROB 5	SUE 3

Children of Families

ROB 5	SUE 3	DON 2

Name of Families
GRAY BELL

Names of Families

GRAY ANN TOM ROB SUE	BELL SUE ED DON

The last two examples illustrate the difference between singulars and plurals: **Name of Families** selected the name of each family; **Names of Families** selected all names of each family. The selected data are also grouped according to the specification: **Children** selected a list of all children; **Children of Families** selected a list of all children for each family thus introducing another level of nesting. It is also to possible to have multiple indices for plural nodes. This way particular items can be selected in the order the user wishes:

Mother of Families Bell Gray

SUE 29	ANN 32

```
Children Sue Rob of Family Gray
```

SUE 3	ROB 5

Finally, it is possible to have lists of 2 or more structural names in between the **of**s. The data selected will be organized in the corresponding fashion:

```
Fathers Mothers
```

TOM 40 | ED 34	ANN 32 | SUE 29

```
Father Mother of Families
```

TOM 40 | ANN 32	ED 34 | SUE 29

```
Age of Father Mother of Families Bell Gray
```

34 29 | 40 32

```
Name of Father Mother Children of Families
```

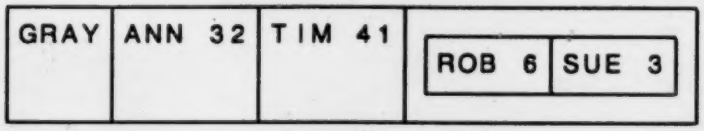
TOM | ANN | ROB SUE	ED | SUE | DON

4.2.3. Modifying Datatalk Data

Just as the accessing of data is analogous to NIAL **reach**, it is possible to directly change any subtree of the data in a way analogous to NIAL **deep-place**. This is the direct mode of modification. For example:

```
Age of Child Rob of Family Gray:= 6
Father of Family Gray:= "TIM 41
```

Family Gray



The following chapter shows how to enter and modify data in the conversational mode.

Chapter 5

DATATALK EXAMPLE

The best way to show how to use Datatalk is through an example. A more formal description of Datatalk is found in Appendix A1 and a reference manual is provided in Appendix A2. An important point to remember when using Datatalk is that all of NIAL is available in addition to the data description, accessing and modification methods discussed in the previous chapter. There are also a number of helpful Datatalk operations that will be seen in the following commented example. Comments are input lines that begin with a number sign (#).

In this example the user, MAY, would like to store a winelist for the *Unheardof* restaurant. The winelist is to be organized by country and within country by red wines and white wines. Having stored it she then proceeds with some report generation, modification, sorting and analysis.

```
Q'Nial: UNIX Release 1 Version 3.03 Dec 6, 1983
clear workspace

      load "Datatalk
      Datatalk

            Welcome to Datatalk

What would you like to be called? May
What is the name of your data? Winelist

What does a WINELIST have? a Name and Countries
Is COUNTRIES a list of similar items? yes

WINELIST:
```

NAME	COUNTRY COUNTRY COUNTRY ...

```
Does this structure look right? yes

What does a COUNTRY have? a Name, Reds and Whites
Is REDS a list of similar items? yes
```

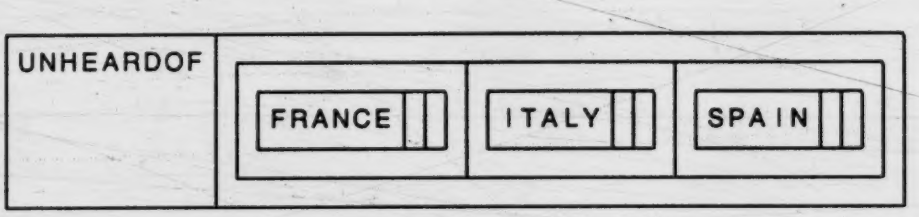
```
Is WHITES a list of similar items? yes

COUNTRY:
```

NAME	RED RED RED ...	WHITE WHITE WHITE ...

```
Does this structure look right? yes

What does a RED have? quit

MAY: # DEFINITION and ENTRY

MAY: # The winelist currently has a
     # name (unknown) and countries (empty list)
     Winelist
```

```
MAY: modify Winelist

WINELIST:
——NAME: ?  Unheardof

Would you like to add a COUNTRY? yes

——COUNTRY 0:
————NAME: ?  France

Would you like to add a RED? stop

Would you like to add a COUNTRY? yes

——COUNTRY 1:
————NAME: ?  Italy

Would you like to add a RED? quit

MAY: # The winelist now has 2 countries each of
     # which has empty lists of reds and whites
     Winelist
```

```
MAY: Name of Winelist

UNHEARDOF

MAY: Name of Countries

FRANCE ITALY

MAY: enter Countries

COUNTRY 2:
——NAME: ?  Spain

Would you like to add a RED? quit

MAY: Winelist
```




```
MAY: # directly define the structure of
     # both red and white wines
     Red White have Name Price Quantity

MAY: show White

WHITE:
NAME PRICE QUANTITY

MAY: # review the structure of the winelist
     review Winelist

WINELIST:
```

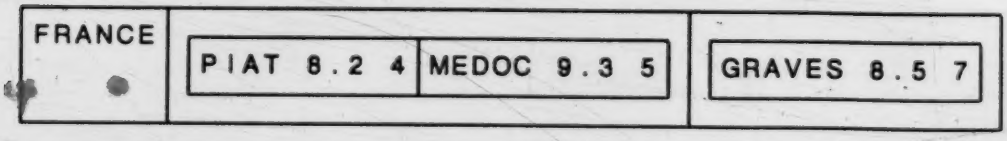
NAME	COUNTRY COUNTRY COUNTRY . . .

```
COUNTRY:
```

NAME	RED RED RED . . .	WHITE WHITE WHITE . . .

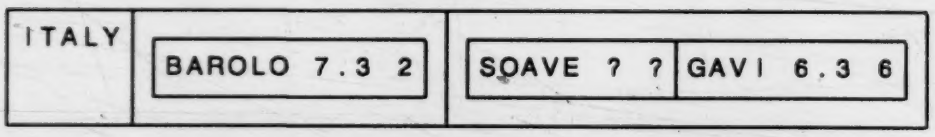
```
RED WHITE:
NAME PRICE QUANTITY

NAME PRICE QUANTITY:
leaf name
```

```
MAY: # DATA MODIFICATION

MAY: # when modifying (or entering) data the
     # user is prompted with the current value
     # (which is ? if unknown). In response
     # the user has 4 choices:
     #  1. hit return - leave as is
     #  2. type in new value
     #  3. stop - stop at that level, go back
     #  4. quit - quit the program
     modify Country France

COUNTRY:
——NAME: FRANCE

Would you like to add a RED? yes

——RED 0:
————NAME: ?  Piat
————PRICE: ?  8.20
————QUANTITY: ?  4

Would you like to add a RED? yes

——RED 1:
————NAME: ?  Medoc
————PRICE: ?  9.30
————QUANTITY: ?  5

Would you like to add a RED? no

Would you like to add a WHITE? yes

——WHITE 0:
————NAME: ?  Graves
————PRICE: ?  8.50
————QUANTITY: ?  7

Would you like to add a WHITE? no

MAY: Country France
```



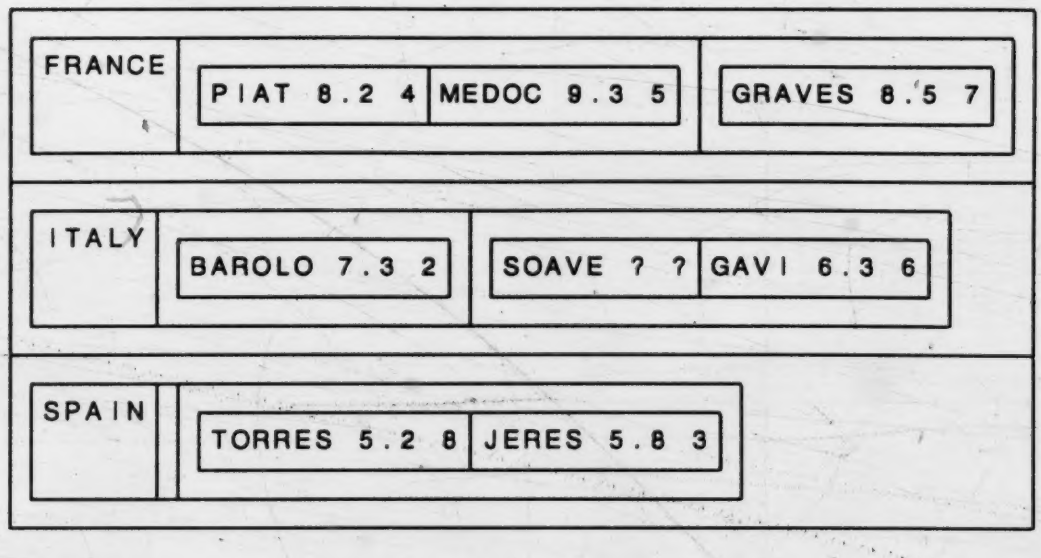

```
MAY: modify Country Italy

COUNTRY:
——NAME: ITALY
```

```
Would you like to add a RED? yes

——RED 0:
————NAME: ?  Barolo
————PRICE: ?  7.30
————QUANTITY: ?  2

Would you like to add a RED? no

Would you like to add a WHITE? yes

——WHITE 0:
————NAME: ?  Soave
————PRICE: ?  stop

Would you like to add a WHITE? yes

——WHITE 1:
————NAME: ?  Gavi
————PRICE: ?  6.30
————QUANTITY: ?  6

Would you like to add a WHITE? no

MAY: Country Italy
```




```
MAY: enter Whites of Country Spain

WHITE 0:
——NAME: ?  Torres
——PRICE: ?  5.20
——QUANTITY: ?  8

Would you like to add a WHITE? yes

WHITE 1:
——NAME: ?  Jeres
——PRICE: ?  5.80
——QUANTITY: ?  3

Would you like to add a WHITE? no
```

```
MAY: # display countries in a column (post)
     post Countries
```




```
MAY: Name of Reds Whites
```

PIAT MEDOC BAROLO	GRAVES SOAVE GAVI TORRES JERES

```
MAY: # SIMPLE ANALYSIS
MAY: EACH tally Reds Whites
3 5
MAY: EACH EACH tally Reds Whites of Countries
```

2 1	1 2	0 2

```
MAY: average is div[sum,tally]
MAY: Price of Reds
8.2 9.3 7.3
MAY: average Price of Reds
8.266666667
```

```
MAY: Price of Whites

8.5 ? 6.3 5.2 5.8

MAY: # one of the prices is unknown
     average Price of Whites

?

MAY: # ignore unknown prices
     average. Price of Whites except ??

6.45

MAY: EACH mix Whites Reds
```

GRAVES	8.5	7	PIAT	8.2	4
SOAVE	?	?	MEDOC	9.3	5
GAVI	6.3	6	BAROLO	7.3	2
TORRES	5.2	8			
JERES	5.8	3			

```
MAY: # direct modification
     Price of White Soave of Country Italy:= 6.10

MAY: # assisted modification
     modify White Soave of Country Italy

WHITE:
——NAME: SOAVE
——PRICE: 6.1
——QUANTITY: ?  1

MAY: sum Quantities

36

MAY: # REPORT GENERATION

MAY: mix. "Name "Price "Quantity hitch Whites

Name    Price Quantity
GRAVES    8.5        7
SOAVE     6.1        1
GAVI      6.3        6
TORRES    5.2        8
JERES     5.8        3

MAY: # define the above labelling operation
     label is mix hitch
```

```
MAY: "Name "Quantity label Name Quantity of Reds

Name    Quantity
PIAT           4
MEDOC          5
BAROLO         2

MAY: Head:= "Name "Price "Quantity

Name Price Quantity

MAY: Data:= EACH Head label Reds Whites
```

Name	Price	Quantity
PIAT	8.2	4
MEDOC	9.3	5
BAROLO	7.3	2

Name	Price	Quantity
GRAVES	8.5	7
SOAVE	6.1	1
GAVI	6.3	6
TORRES	5.2	8
JERES	5.8	3

```
MAY: mix. "RedWine "WhiteWine pair Data
```

RedWine		
Name	Price	Quantity
PIAT	8.2	4
MEDOC	9.3	5
BAROLO	7.3	2

WhiteWine		
Name	Price	Quantity
GRAVES	8.5	7
SOAVE	6.1	1
GAVI	6.3	6
TORRES	5.2	8
JERES	5.8	3

```
MAY: # define the above tabling operation
     table is mix pair

MAY: H0:=Name of Countries

FRANCE ITALY SPAIN

MAY: H1:="Name "Cost "Qty

Name Cost Qty
```

```
MAY: # combine table and label to produce report
     H0 table EACH H1 label Whites of Countries
```

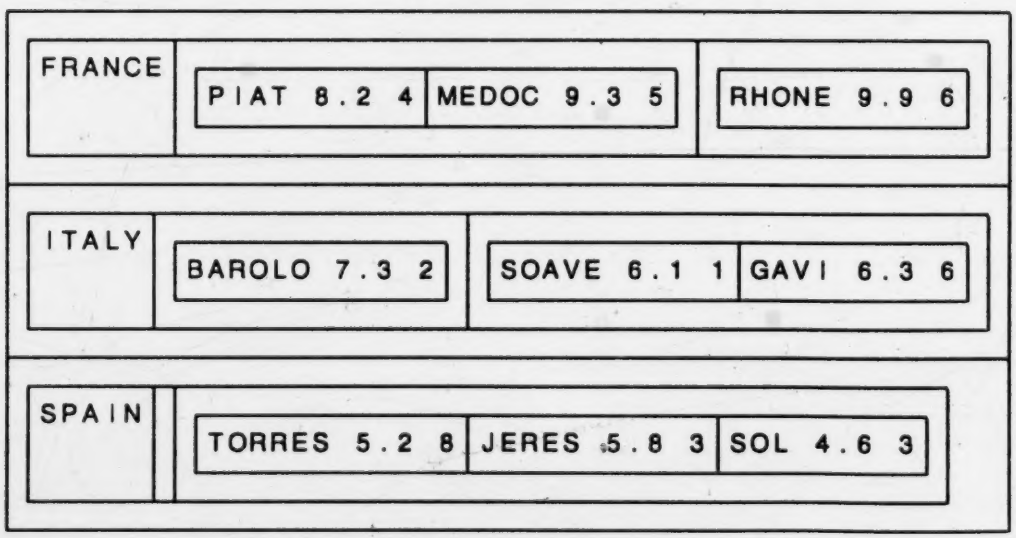
FRANCE			ITALY			SPAIN		
Name	Cost	Qty	Name	Cost	Qty	Name	Cost	Qty
GRAVES	8.5	7	SOAVE	6.1	1	TORRES	5.2	8
			GAVI	6.3	6	JERES	5.8	3

```
MAY: # define a report which will always
     # reflect the current data
     Report is \
     H0 table EACH H1 label Whites of Countries

MAY: modify Countries

COUNTRY 0:
——NAME: FRANCE

——RED 0:
————NAME: PIAT  stop

——RED 1:
————NAME: MEDOC  stop

Would you like to add a RED? no

——WHITE 0:
————NAME: GRAVES  stop

Would you like to add a WHITE? yes

——WHITE 1:
————NAME: ?  Rhone
————PRICE: ?  9.90
————QUANTITY: ?  6

Would you like to add a WHITE? no

COUNTRY 1:
——NAME: ITALY  stop

COUNTRY 2:
——NAME: SPAIN

Would you like to add a RED? no

——WHITE 0:
————NAME: TORRES  stop

——WHITE 1:
```

```
———NAME: JERES  stop

Would you like to add a WHITE? yes

—WHITE 2:
———NAME: ?  Sol
———PRICE: ?  4.60
———QUANTITY: ?  3

Would you like to add a WHITE? no

Would you like to add a COUNTRY? no

MAY: Report
```

FRANCE			ITALY			SPAIN		
Name	Cost	Qty	Name	Cost	Qty	Name	Cost	Qty
GRAVES	8.5	7	SOAVE	6.1	1	TORRES	5.2	8
RHONE	9.9	6	GAVI	6.3	6	JERES	5.8	3
						SOL	4.6	3

```
MAY: # DATA DELETION
     delete White Graves of Country France

MAY: post Countries
```




```
MAY: # DATA SORTING

MAY: sort_by is choose[gradeup second,first]
```

```
MAY: Head label. Whites sort_by Price of Whites

Name    Price Quantity
SOL       4.6        3
TORRES    5.2        8
JERES     5.8        3
SOAVE     6.1        1
GAVI      6.3        6
RHONE     9.9        6

MAY: Head label. Whites sort_by Name of Whites

Name    Price Quantity
GAVI      6.3        6
JERES     5.8        3
RHONE     9.9        6
SOAVE     6.1        1
SOL       4.6        3
TORRES    5.2        8

MAY: # DATA SELECTION

MAY: Price of Whites > 6 sublist Name of Whites

RHONE SOAVE GAVI

MAY: where is CONVERSE sublist

MAY: mix. Whites where. Quantity of Whites <= 3

SOAVE 6.1 1
JERES 5.8 3
SOL   4.6 3

MAY: Head label. Reds sort_by Quantity of Reds

Name    Price Quantity
BAROLO    7.3        2
PIAT      8.2        4
MEDOC     9.3        5

MAY: show Red

RED:
NAME PRICE QUANTITY

MAY: mix Reds

PIAT   8.2 4
MEDOC  9.3 5
BAROLO 7.3 2
```

```
MAY: # The structure (and thus the data) can
     # be redefined or rearranged at any time
     Red has Name Quantity Sweetness Price

MAY: mix Reds

PIAT   4 ? 8.2
MEDOC  5 ? 9.3
BAROLO 2 ? 7.3

MAY: "Name "Qty "Sweetness "Price label Reds

Name   Qty Sweetness Price
PIAT     4 ?           8.2
MEDOC    5 ?           9.3
BAROLO   2 ?           7.3

MAY: review Winelist

WINELIST:
```

NAME	COUNTRY COUNTRY COUNTRY ...

```
COUNTRY:
```

NAME	RED RED RED ...	WHITE WHITE WHITE ...

```
RED:
NAME QUANTITY SWEETNESS PRICE

WHITE:
NAME PRICE QUANTITY

NAME QUANTITY SWEETNESS PRICE:
leaf name

MAY: stop

     bye
```

Datatalk data can be saved through NIAL's workspace mechanism. Datatalk will resume working with the same data when next called. A description of some analytical, searching and report generating utilities that were found useful are in Appendix A3.

Chapter 6

CONCLUSION

This thesis shows how it is possible to provide a powerful and versatile yet simple system for the storage and analysis of data which is suitable for use by non-expert programmers. A mechanism for data management has been presented as an integrated part of a programming language in an interactive environment. In this environment a user can easily expand his knowledge of the data and its analysis at his own pace.

Datatalk employs a new way of describing and accessing data which greatly simplifies these activities. It allows the user to describe his data and its structure informally and to access the data with the structure names used in its description. The amount of information required from the user to specify the relationships among the data is minimized. In addition, Datatalk has not erected any barriers between the user and the underlying power of the simple very high-level language, NIAL. Datatalk combined with NIAL is a powerful and practical tool for programming in the small.

The work done on Datatalk with NIAL has encouraged some observations and suggested some possible extensions.

6.1. Observations

The basic observation is that a good way to provide an integration of user-friendly data management with useful analysis is to embed the management in an easy-to-use very high-level language.

So-called "user-friendly" systems are often largely or entirely prompt driven and rarely incorporate a full programming language. This tendency is

due to an underestimation of user intelligence or an assessment (usually correct) of the underlying language as being too cumbersome. In developing this thesis work it was found that the less NIAL was covered up the simpler and more powerful the system became. This led to the incorporation of Datatalk into NIAL as an extension to the language. The user is not restricted to some subset of operations provided by the designer through an "interface". In this sense Datatalk may be called an "open" system as opposed to a "closed" one. This sort of openness is necessary to the power and versatility of a system.

Another observation is that a sound theoretical base is essential. C. J. Date says it well in An Introduction to Database Systems: [Dat81]

> "[the system's] behaviour must be totally predictable and, to the greatest extent possible, should accord with users' intuitive expectations. Surprises, especially unpleasant ones, simply cannot be tolerated. Whatever formal system we choose as a basis for the conceptual level, we *must* be fully aware of exactly what is and is not possible in that system. Specifically, we should be certain that ambiguity and paradox cannot occur. In short, we should know exactly what we are doing."

A guiding principle in the design of Datatalk was to make the syntax consistent with that of NIAL. Both Datatalk and NIAL are founded on precise concepts. Although the statements of both are very English-like (and thus easy to create and understand), the underlying rules of syntax and semantics are simple, small and above all - precise. Datatalk and NIAL never guess what users might want. They provide an environment where it is easy for users to say exactly what they do want.

This leads to observation on 'natural language'. Terry Winograd concludes in his article Computer Software for Working with Language: [Win84]

> "Often it is taken for granted that natural language is the best way for people to communicate with computers. ... It is not at all evident, however, that the proposition is valid. In some cases even the fullest understanding of natural language is not as expressive as a picture. And in many cases a partial understanding of natural language proves to be less usable than a well-designed formal interface."

In presenting a "natural language" understanding front-end to the user a false impression of computer intelligence and second-guessing may be fostered. The user comes to mistrust the program's ability to correctly interpret his statements. In such a situation the user soon develops a desire to determine a formal understructure that he can employ with confidence.

The approach in Datatalk has been to provide some guidance at the beginning of a session and at any point the user desires. The transition to the direct use of the formal Datatalk and NIAL notation is natural and comfortable.

The final observation is that NIAL is a good language for implementing interactive software systems, and is especially well-suited for those to be used by nonprogrammers. It is also an excellent language for experimenting with data models and database systems.

6.2. Extensions

An interesting extension to Datatalk or a system like Datatalk would be a visual programming capability, that is, to be able to define data structures and modify and retrieve data with a visual interface.

Both the template tree and the corresponding data tree might be on a screen. To "show" what the data structure is rather than "tell" what it is, the user might join structures together with a mouse, creating an aggregation

(superstructure). These structures (nodes in the template tree) might be labelled with names or icons. In order to access any particular subtree of the data for the purpose of modification or retrieval, the user could point to the subtree of interest. In order to select all subtrees of certain type(s) a pointing mechanism utilizing the appropriate node(s) of the template tree could be used.

Another possible extension would be to allow tables (and higher dimensional arrays). The Datatalk Template and Data are trees based on lists of lists. NIAL is actually more general than this. NIAL data structures are arrays of arrays. Thus the idea of singulars (*aggregations*) and plurals (open-ended *collections* of singulars) would still hold but the singulars need not be just lists. They could be tables. To draw this nicely 3-dimensions would be needed. With the direct modification (assignment) capability it is already possible for the *leaves* of the tree to be any NIAL object. This extension would allow a more complex structuring at higher levels in the tree.

References

[Ada83] W. S. Adams, Plain Programming in Nial, Tech. Rep.-83-11, University of Alberta, Edmonton, Canada , 1983.

[Ada84] W. S. Adams, Towards Natural Language Programming, *Proceedings of Canadian Information Processing Society (Session 84)*, May 1984, 17-24.

[Boo51] G. Boole, *An Investigation of the Laws of Thought*, Dover Publications, New York, 1951. Originally published in 1954 by Walton and Maberly, London and by MacMillan and Co., Cambridge. Also available in Volume II of the Collected Logical Works of George Boole, Open Court Publishing Co., La Salle, Illinois, 1916.

[BoS81] W. G. Bouricius and N. R. Sorensen, An Informal Introduction to Array Theory With Applications to a Language and a Database, IBM Cambridge Scientific Center Technical Report No. G320-2135, Cambridge, Mass., USA, May 1981.

[Cod82] E. F. Codd, Relational Database: A Practical Foundation for Productivity, *Comm. ACM 25*, 2 (Feb. 1982), 109-117. (1981 ACM Turing Award Lecture).

[Dat81] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, third edition 1981.

[DSS83] R. B. K. Dewar, E. Schonberg and J. T. Schwartz, *High-level Programming - An Introduction to the Programming Language SETL*, Courant Institute of Mathematical Sciences, New York, 1983.

[HHK77] M. Hammer, W. G. Howe, V. J. Kruskal and I. Wladawsky, A Very High Level Programming Language for Data Processing Applications, *Comm. ACM 20*, 11 (Nov. 1977), 832-840.

[Hen84] L. J. Hendren, ISON - An Introductory Subset of Nial, Technical Report #84-156, Queen's University, Kingston, Canada, 1984.

[Ive62] K. E. Iverson, *A Programming Language*, John Wiley and Sons Inc., New York, London, Sydney, 1962.

[Ive80] K. E. Iverson, Notation as a Tool of Thought, *Comm. ACM 23*, 8 (Aug. 1980), 444-465. (1979 ACM Turing Award Lecture).

[Jen82] M. A. Jenkins, *The Q'Nial Reference Manual*, Computing and Information Science, Queen's University, Kingston, Canada, Mar. 1982. (Version 1.0).

[Jen84] M. A. Jenkins, *A Tutorial Introduction to Nial*, Queen's University, Kingston, Canada, 1984.

[Mar82] J. Martin, *Application Programming Without Programmers*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[Mas84] F. A. Masterson, Languages For Students, *BYTE 9*, 6 (June 1984), 233-238.

[MAE65] J. McCarthy, P. W, Abrahams, D. J. Edwards, T. P. Hart and M. I. Levin, *LISP 1.5 Programmer's Manual* , The MIT Press, Cambridge, Mass., 1965.

[Mor79] T. More, The Nested Rectangular Array as a Model of Data, *APL79, APL Quote Quad 9*, 4 (June 1979), 55-73.

[Mor81] T. More, Notes on the Diagrams, Logic and Operations of Array Theory, IBM Cambridge Scientific Center Technical Report No. G320-2137, Cambridge, Mass., USA, Sep. 1981.

[Oll80] T. W. Olle, Data Base Management Systems, in *Programming Language Standardisation*, vol. 7, I. D. Hill and B. L. Meek (ed.), Ellis Horwood Limited, Chichester, England, 1980, 107-117.

[Rei81] P. Reisner, Human Factors Studies of Database Query Languages: A Survey and Assessment, *Computing Surveys 13*, 1 (Mar. 1981), 13-31.

[RoF83] J. Rockart and L. Flannery, The Management of End User Computing, *Comm. ACM 26*, 10 (Oct. 1983), 776-784.

[ScJ82a] Fl. Schmidt and M. A. Jenkins, Systems Design and the Nial Approach, (Queen's University, Kingston, Canada), June 1982.

[ScJ82b] Fl. Schmidt and M. A. Jenkins, Array Diagrams and the Nial Approach, *APL82, APL Quote Quad 13*, 1 (1982), 315-319.

[Sha84] M. Shaw, Abstraction Techniques in Modern Programming Languages, *IEEE Software 1*, 4 (Oct. 1984), 11-26.

[SmS77] J. M. Smith and D. C. P. Smith, Database Abstractions: Aggregation, *Comm. ACM 20*, 6 (June 1977), 405-413.

[Sut83] L. Sutherland, The Friendly Q'Nial Users Guide for Berkeley 4.1 bsd Unix, (Final Draft) , 1983.

[WOS83] T. Watanabe, T. Ohsawa and T. Suzuki, A Simple Database Language for Personal Computers, *Comm. ACM 26*, 9 (Sep. 1983), 646-653.

[Win79] T. Winograd, Beyond Programming Languages, *Comm. ACM 22*, 7 (July 1979), 391-401.

[Win84] T. Winograd, Computer Software for Working with Language, *Scientific American 251*, 3 (Sep. 1984), 130-145.

[Zld77] M. M. Zloof and S. P. de Jong, The System for Business Automation (SBA): Programming Language, *Comm. ACM 20*, 6 (June 1977), 385-395.

Appendix A1

DATATALK TERMINOLOGY

Explanation is in Roman.
Definitions are in Italics.
Examples are in Helvetica.

A set of structural definitions introduces *names*.
These names are associated with the nodes of a *tree*.
Collections of subtrees can be reached with *paths*.
Single subtrees can be reached with *spaths*.

e.g.

Winelist has Name Countries
Country has Name Reds Whites
Red White have Name Price Quantity

name: Structure definitions are made up of singulars and plurals.
The singulars correspond to single items.
The plurals correspond to variable length lists.
All *names* have singular and plural forms.

sname	- singular	**Name Price Quantity**
pname	- plural	**Countries Reds Whites**
psname	- plural of *sname*	**Names Prices Quantities**
spname	- singular of *pname*	**Country Red White**

tree: The structure template is generated by the definitions.
Nodes are either single (*snode*) or plural (*pnode*).
This is a template because the *pnodes* have variable length.

snode
- generated by *sname*
- sons are *snodes* and *pnodes*
- or is a leaf (i.e. holds data)

pnode
- generated by *pname*
- sons are similar *snodes*

path: To select a subtree or lists (of lists (...)) of subtrees.
The nesting arrangement of the result corresponds to the request.
Singulars pick **the** subtree having that structure.
Plurals get a list of **all** subtrees having that structure.

sname	- pick *snode*	**Price**
spname index	- pick *snode* of *pnode*	**Country Spain**
psname	- get all subtrees	**Prices**
pname	- get all subtrees	**Countries**
pname indexs	- get *snodes* of *pnode*	**Reds Gavi Piat**
names	- list of the above	**Reds Name Prices**
path of *path*	- nested selection	**Price of Whites Reds**

spath: To reach a specific subtree.
i.e. A *path* which picks a single subtree.

sname	**Name**
spname index	**Country Spain**
spath of *spath*	**Name of Country 2**

index: An *index* is used when accessing the son(s) of *pnodes*.
Name is a possible index when it is an attribute of the sons.
i.e. **Name** acts as an automatic key.
0-origin position integer(s) can always be used.

e.g.

Country France
Reds Moreau Graves
Country 1
Whites 2 0 3

Appendix A2

REFERENCE MANUAL

2.1. Overview

This section contains a list of Datatalk operations dealing with the description, modification and retrieval of the user's data. For description and modification there is a direct mode and an assisted (conversational - prompted) mode. For a list of Datatalk terminology see Appendix A1. An extended example is in Chapter 5.

Description

Direct:	*sname* **has** *name*(s)	- set aggregation for name
	snames **have** *names*(s)	- set aggregation for names
Assisted:	**define** *sname*	- define aggregation
	plan *sname*	- define, then define sub-aggregations
Review:	**show** *sname*	- show aggregation
	review *sname*	- show; then show sub-aggregations

Modification

Direct:	*spath* := ...	- set subtree with value of ...
Assisted:	**modify** *spath*	- modify selected subtree
	enter *spath*	- add new son to *pnode*
	delete *spath*	- delete one son of *pnode*

Retrieval

... *path* ...	- select corresponding subtree(s)

Other

correct *old new*	- correct the spelling of a name
help *operation*	- help on specified operation

2.2. General Notes

1. All input must be followed by **return**.

2. When starting a new Datatalk session the user is asked for:
   The name of the user. This becomes the Datatalk prompt.
   The name of the data. This is then immediately **plan**ned.

3. Any NIAL expression may be entered in response to the Datatalk prompt.

4. The following Datatalk operations are assisted (i.e. have prompts):
   plan define - aggregation description
   modify enter delete - data modification

5. **Stop** or **Quit** can be entered in response to any prompt.
   Stop will stop and go back one level.
   Quit will quit the current program.
   e.g. Stop or Quit will leave the Datatalk prompt.

2.3. Data Description

Direct:	*sname* **has** *name*(s)	**Country has Name Reds Whites**
	snames **have** *name*(s)	**Red White have Price Name**
Assisted:	**define** *sname*	**define Country**
	plan *sname*	**plan Country**

The data description defines the Datatalk Template. The Datatalk Data Tree (which holds the user's data) always corresponds to this template. Aggregations can be defined (and redefined) at any time. In the case of new definitions unknown values (**??**) are put in at new leaf nodes. In the case of redefinition the Data Tree is reorganized to reflect the change of template. For example, suppose **Red has Name Price** originally and is later changed to **Red has Price Quantity Name**. All **Red** nodes in the Data Tree will reorganize accordingly - **Name** and **Price** will be switched around and **??** (for the new unknown quantity) will be placed in between. New *pnodes* in the Tree start off being empty.

The responses to the assisted definitions (**define** and **plan**) may include noise words like 'a' and 'an' which are ignored. Commas and 'and's act as separators.

Review:	**show** *sname*	**show Country**
	review *sname*	**review Country**

These two operations are for reviewing the structure of the data. Plural nodes are depicted as having an open ended list of sons.

2.4. Data Modification

Direct:	*spath* := ...	**Name of Country 1:= "SPAIN** **Whites of Country Spain:= ...**
Assisted:	**modify** *spath*	**modify Red Plat of Country France**
	enter *spath*	**enter Reds of Country Spain**
	delete *spath*	**delete White Gavi of Country Italy**

In direct mode, the value of any Datatalk (including NIAL) expression can be placed in the Tree by using NIAL assignment. This is especially useful for large amounts of data. Of course, the data must be structured to fit the template of the specified subtree.

Modify can be applied to any subtree. It reaches down to the leaves to allow the entry of data. The user is prompted with the current value of the leaf (which may be **??**). There are four possible responses:

1. Type in new value. This is made up of number(s) and phrase(s).
   If more than one value is entered it becomes a list.
   The Tree immediately incorporates the change.
2. Hit **return**. This will leave the leaf value as is.
3. Type **Stop**. This will stop and go back a level.

4. Type **Quit**. This will quit **modify**.

New subtrees can only be **entered** as sons of *pnodes*. Similarly it is only possible to **delete** sons of *pnodes*. After adding a new node **enter** behaves like **modify**.

2.5. Data Retrieval

... *path* ...	**Price Quantity of Red Moreau**

Data is retrieved by naming the item or items of interest. If a plural is used, the result is a list of all data items described by the corresponding singular, grouped according to any *names* specified in the path description that follows. When *paths* are used this way in a NIAL expression, they are replaced by the corresponding data before any evaluation takes place.

See Appendix A1 and Chapter 5 for more detail and examples.

2.6. Other

correct *old new*	**correct Brothers Brethren**
help "*operation*	**help "modify**

Correct will change all occurrences of the old spelling with the new. **Help** will cause a help message to be displayed on the specified subject.

Appendix A3

DATATALK UTILITIES

3.1. Reporting

NIAL: Builders and Permuters - e.g. **post mix flip hitch append link**

label is mix hitch *Labels* **label** *List*

- label a list of items with identical structure
e.g. **"Name "Price label Name Price of Whites**

table is mix pair *Titles* **table** *List*

- label a list of diverse items
e.g. **"Redwine "Whitewine table Name of Reds Whites**

table1 is table[first,EACH mix second] *Titles* **table1** *List*

- table for lists with more complex items
e.g. **"Redwine "Whitewine table Reds Whites**

3.2. Searching/Selecting

NIAL: Predicates and Selectors - e.g. **< = sublist choose**

where is CONVERSE sublist *List* **where.** *Condition*

- select items of list where condition is true.
e.g. **Name Price of Whites where. Quantity of Whites > 5**
e.g. **Countries where. EACH tally Whites of Countries < 4**

3.3. Sorting

NIAL: Sort primitives - e.g. **gradeup gradedown grade sort**

sort_by is choose[gradeup second,first] *List* **sort_by** *X*

- sort list by gradeup (ascending order) of sorter X.
e.g. **Name Quantity Price of Reds sort_by Price of Reds**

down_by is choose[gradedown second,first] *List* **down_by** *X*

- sort list by gradedown (descending order) of sorter X.
e.g. **Countries down_by EACH sum Quantities of Countries**

3.4. Analysis

NIAL: Arithmetic and Measurements - e.g. **sum tally**

average is div[sum,tally] **average** *List*

median is pick[floor. .5*tally,"< sort] **median** *List*

Appendix A4

DATATALK PROGRAMS

```
Dp:=Dd:=Sn:=Pn:=??; Lpn:=Null; Stop:=Quit:=o; Td:=Ts:=Ti:=Null;

R0:="MODIFY "DELETE "ENTER "CORRECT;
R1:="SHOW "REVIEW "DEFINE "PLAN "AID "CLEAR;
R2:="HAS "HAVE;
R3:=":= "GETS;

Ra:="YES "Y "OK;
Rl:="NAME (fault '?invalid name');
Rn:=(phrase ',) "AND "A "AN;
Rp:="CHILDREN "GEESE;
Rs:="CHILD    "GOOSE;

Ds is first Sn; Sqs is {LOCAL;Stop:=Quit:=o;}; Blnk is writescreen ' ';

ni is CONVERSE in; num is 0 0. ni type;

spn is link Sn Pn "OF ni; sk is link EACH sketch;

nam is and[not spn,'?u'=. 2 take string execute];

ni is CONVERSE in; num is 0 0. ni type;

ECHALL is TR f op D{IF or EACH empty D THEN null
  ELSE EACHALL f D ENDIF}

pl is op N{N find Sn pick Pn}; sl is op N{N find Pn pick Sn}
```

```
pln is or[Rp ni,and['S=last,'SS'~=. 2 takeright]string];

pz is phrase op N{V:=2 takeright N;
 IF last V='Y and. first V notin 'AEOU'
  THEN front N link 'IES'
  ELSE IF last V='S or. last V='H and. first V in 'CS'
   THEN N link 'ES'
   ELSE N append 'S ENDIF ENDIF} string;

sz is op N{IF N in Rp THEN Rs@(N find Rp) ELSE N:=string N;
 I:= 3 takeright N find 'SES' 'IES' 'HES';
 phrase. I pick 2 3 2 1 dropright N link. I pick "'Y ENDIF}

slr is FORK[Pn ni,sl,pass]; cll is EACH slr cull content EACH value;

wds is op P{T:=readscreen sk P; IF T allin ' THEN Null ELSE
 T:=EACH FORK[num third,third,first] second scan T;
 IF first T in "STOP "QUIT
  THEN first T assign 1;Null
  ELSE T ENDIF ENDIF}

ana is FORK['AEIOU' ni first,'an ' link,'a ' link]string;

ph is EACH FORK[num,pass,phrase. '" link string];
```

```
prt is op T{T:=ph T;T#(""OF findall T):=',;execute sk '[ T ']}

spl is op N T{I:=1 find EACHLEFT in T N;[I take,I+1 drop]T}

fix is op I T{IF empty T THEN IF I THEN ') ELSE Null ENDIF ELSE
 J:=or[spn,I and or[num,nam]] first T;
 I:=(J and not I)(I and not J)sublist EACH phrase '(get' ');
 link I(FORK[J first,ph,pass]first T)(J fix rest T) ENDIF}

dex is execute sketch. o fix;

si is op S N{N find value S}

pi is op D S I{
 IF nam first I
  THEN I:=I EACHLEFT find. S si "NAME EACHRIGHT pick D ENDIF;
 IF tally D in I THEN'?? ELSE I ENDIF}

pth is op D S P{IF empty P THEN Null ELSE
 P N:=[front,last] P;
 IF atomic N
  THEN IF N in value S THEN I:=S si N ELSE I:=?? ENDIF
  ELSE N I:=N; J:=S si pl N; I:=J pair pi D@J N I ENDIF;
 IF ?? in I THEN ??path ELSE I link pth D@@I N P ENDIF ENDIF}
```

```
lgt is op D S N{
 IF N=S
  THEN solitary D
  ELSE IF S in Lpn
      THEN link ECHALL lgt D (sl S) N
      ELSE IF atomic(S:=value S)
          THEN Null
          ELSE link EACHALL lgt D S N
         ENDIF ENDIF ENDIF}

gt is op D S P{IF empty P THEN D ELSE P N:=[front,last] P;
 IF atomic N
  THEN IF N in Sn
     THEN IF N in value S
          THEN gt D@(S si N) N P ELSE ??path ENDIF
     ELSE ECHALL gt (lgt D S(sl N))(sl N)(single P) ENDIF
  ELSE IF or[num,nam] last N
      THEN N I:=[slr first,rest] N; J:=S si pl N;
         IF ?? =(I:=pi D@J N I) THEN ??path
          ELSE IF 1=tally I
              THEN gt(I pick D@J)N P
              ELSE EACHALL gt(I choose D@J)N(single P)
         ENDIF ENDIF
      ELSE EACHALL gt(single D)S(P EACHRIGHT append N)
    ENDIF ENDIF ENDIF}
```

```
typ is op S{IF S in Lpn THEN Null ELSE S:=value S;
 IF atomic S THEN ?? ELSE EACH typ S ENDIF ENDIF}

tch is Dd Ds append FORK[Ds=last,front,pass] prt;

chk is op V{IF ??path notin content V THEN V
 ELSE writescreen '** invalid path **'; ENDIF}

path is chk pth tch; get is chk gt tch;

suit is FORK[1=tally,first,pass]; mdy is op D S I{o}

ntr is op S J I{IF Stop or Quit THEN Null ELSE Blnk;
 IF "YES in wds 'Would you like to add '(ana S)'? '
  THEN hitch(mdy(typ S)(S J)I)(ntr S(J+1)I)
  ELSE Null ENDIF ENDIF}

mdy is op D S I{LOCAL V J;IF Stop or Quit THEN D ELSE
 IF S in Lpn
  THEN S:=sl S; J:=tally D;
     D:=ECHALL mdy D(S EACHRIGHT append tell J)I;
     D link (ntr S J I)
  ELSE IF atomic(V:=value first S)
      THEN T:=suit wds (I reshape '-) S ': ' D '  ';
         IF empty T THEN D ELSE T ENDIF
      ELSE Blnk; writescreen sk (I reshape '-) S ':;
         D:=EACHALL mdy D V(I+2); Stop:=o; D
     ENDIF ENDIF ENDIF}
```

```
modify is op T{LOCAL V; IF ?? ~=type(V:=path T) THEN
 Dd:=(mdy (V reach Dd)(first T)0) V deepplace Dd ENDIF}

enter is op T{IF ?? ~=type(V:=path T) THEN
 D:=Dd@@V; S:=sl first T; J:=tally D;
 Dd@@V:=D link hitch(mdy(typ S)(S J)0)(ntr S(J+1)0) ENDIF}

delete is op T{IF ?? ~=type(V:=path T) THEN
 V I:=[front,last]V; D:=Dd@@V;
 Dd@@V:=not(I match tell tally D) sublist D ENDIF}

shw is EACH FORK[Pn ni,"... CONVERSE append. 3 reshape sl,pass];

show is op S{Blnk; writescreen sk S ':; S:=value S;
 IF atomic S THEN writescreen 'leaf name' ELSE write shw S ENDIF;}

review is op S{N:=Rl;
 REPEAT EACH show S; S:=cll S except(N:=N link S)
 UNTIL empty S ENDREPEAT;}
```

```
fll is op D S{
 IF S in Lpn
  THEN D ECHALL fll sl S
  ELSE IF S=Ts
      THEN D Ti placeall Td
      ELSE S:=value S;
         IF atomic S
          THEN D
          ELSE EACHALL fll D S
         ENDIF ENDIF ENDIF}

clear is op S{LOCAL P V; Ts:=S; Td:=??; Ti:=Null; Dd:=fll Dd Ds;
 V:=EACH slr value S; erase S; S:=V except cll Sn;
 Lpn:=Lpn except(P:=EACH pl S); Pn:=Pn except P; Sn:=Sn except S;}

hs is op B S N{LOCAL P; Lpn:=Lpn link(P:=B sublist N except Pn);
 Ts:=N; Td:=typ "Ts; Ts:=S; Ti:=value S EACHLEFT find N;
 Dd:=fll Dd Ds; S assign N; S:=not B sublist N except Sn;
 Pn:=link Pn P(EACH pz S); Sn:=link Sn(EACH sz P)S;}

has is hs[EACH pln last,first,last];
```

```
pck is op N{IF not. Quit or Stop THEN
 IF N in Pn THEN l
 ELSE IF N in Sn THEN o
     ELSE IF pln N
         THEN first wds 'Is 'N' a list of similar items? ' in Ra
         ELSE o ENDIF ENDIF ENDIF ENDIF}

define is op S{LOCAL N B; IF not Quit THEN Stop:=o;
 REPEAT Blnk; N:=wds 'What does '(ana S)' have? ' except Rn;
 IF not empty N
  THEN B:=EACH pck N;
    IF not. Quit or Stop
     THEN hs B S N; show S;
         IF first wds 'Does this structure look right? ' in Ra
          THEN Stop:=l
          ELSE clear S ENDIF ENDIF ENDIF
 UNTIL Stop or Quit or empty N ENDREPEAT ENDIF}

plan is op S{N:=Rl link sublist[EACH. not atomic value,pass] Sn;
 REPEAT EACH define S; S:=cll S except(N:=N link S);
 UNTIL Quit or empty S ENDREPEAT}
```

```
aid is op N{write N;}

rpl is op OldNew V{(last OldNew)(first OldNew find V)place V}

correct is op OldNew{LOCAL;
 Sn:=OldNew rpl Sn; Pn:=OldNew rpl Pn; Lpn:=OldNew rpl Lpn;
 EACH(assign[pass,OldNew rpl value])Sn;
 write sk (first OldNew) ' is now ' (last OldNew);}

ct is suit. "PTH "NAM "HAS "GET CONVERSE sublist
 [R0 ni first,
  R1 ni first,
  or. R2 EACHLEFT in,
  and[spn first,or. R3 EACHLEFT in]];
```

```
Datatalk is{LOCAL T P; Sqs; Blnk;
 writescreen '       Welcome to Datatalk'; Blnk;
 IF Dp=?? THEN Dp:=wds 'What would you like to be called? ' ENDIF;
 IF Ds=?? THEN Sn:=first wds 'What is the name of your data? ';
          Pn:=??; Lpn:=Null; plan Ds; Sqs ENDIF;
 REPEAT Blnk; T:=wds Dp ': '; CASE ct T FROM
  "PTH   : apply[first,rest] T; Sqs END
  "NAM   : EACHRIGHT apply[first,rest] T END
  "HAS   : EACHLEFT has spl R2 T END
  "GET   : P T:=spl R3 T; Dd:=(dex T)(path P) deepplace Dd END
  ELSE  Blnk; FORK[??noexpr =,pass,write] dex T ENDCASE
 UNTIL Stop or Quit ENDREPEAT}
```